**Green Pace Developer: Security Policy Guide Template**



# Green Pace Secure Development Policy

## Contents

[Overview 2](#_Toc52464053)

[Purpose 2](#_Toc52464054)

[Scope 2](#_Toc52464055)

[Module Three Milestone 2](#_Toc52464056)

[Ten Core Security Principles 2](#_Toc52464057)

[C/C++ Ten Coding Standards 3](#_Toc52464058)

[Coding Standard 1 4](#_Toc52464059)

[Coding Standard 2 5](#_Toc52464060)

[Coding Standard 3 6](#_Toc52464061)

[Coding Standard 4 7](#_Toc52464062)

[Coding Standard 5 8](#_Toc52464063)

[Coding Standard 6 9](#_Toc52464064)

[Coding Standard 7 10](#_Toc52464065)

[Coding Standard 8 11](#_Toc52464066)

[Coding Standard 9 13](#_Toc52464067)

[Coding Standard 10 14](#_Toc52464068)

[Defense-in-Depth Illustration 15](#_Toc52464069)

[Project One 15](#_Toc52464070)

[1. Revise the C/C++ Standards 15](#_Toc52464071)

[2. Risk Assessment 15](#_Toc52464072)

[3. Automated Detection 15](#_Toc52464073)

[4. Automation 15](#_Toc52464074)

[5. Summary of Risk Assessments 16](#_Toc52464075)

[6. Create Policies for Encryption and Triple A 16](#_Toc52464076)

[7. Map the Principles 17](#_Toc52464077)

[Audit Controls and Management 18](#_Toc52464078)

[Enforcement 18](#_Toc52464079)

[Exceptions Process 18](#_Toc52464080)

[Distribution 19](#_Toc52464081)

[Policy Change Control 19](#_Toc52464082)

[Policy Version History 19](#_Toc52464083)

[Appendix A Lookups 19](#_Toc52464084)

[Approved C/C++ Language Acronyms 19](#_Toc52464085)

## Overview

Software development at Green Pace requires consistent implementation of secure principles to all developed applications. Consistent approaches and methodologies must be maintained through all policies that are uniformly defined, implemented, governed, and maintained over time.

## Purpose

This policy defines the core security principles; C/C++ coding standards; authorization, authentication, and auditing standards; and data encryption standards. This article explains the differences between policy, standards, principles, and practices (guidelines and procedure): [Understanding the Hierarchy of Principles, Policies, Standards, Procedures, and Guidelines](https://www.linkedin.com/pulse/understanding-hierarchy-principles-policies-standards-wally-beddoe/).

## Scope

This document applies to all staff that create, deploy, or support custom software at Green Pace.

## Module Three Milestone

### Ten Core Security Principles

| **Principles** | Write a short paragraph explaining each of the 10 principles of security. |
| --- | --- |
| 1. ValidateInput Data | Validate input from all untrusted data sources. Proper input validation will eliminate most software vulnerabilities. |
| 1. Heed Compiler Warnings | Compile code using the highest warning level available for your compiler and modify the code to eliminate the warnings. Use static and dynamic tools to mitigate additional security flaws. |
| 1. Architect and Design for Security Policies | Create software architecture and design your software to enforce and implement security. |
| 1. Keep It Simple | Keep the design as small and simple as possible. More complex designs increase the odds of flaws and vulnerabilities. |
| 1. Default Deny | Access permission should be based on permission rather than exclusion. By default, access should be denied but given when required. |
| 1. Adhere to the Principle of Least Privilege | Each process should be completed with the lowest required privilege. Elevated permissions should be provided for the least amount of time necessary to complete the task. |
| 1. Sanitize Data Sent to Other Systems | Sanitize data sent to complex subsystems, such as command shells and relational databases. Unused functionality in these systems may be used for an attack. |
| 1. Practice Defense in Depth | Mitigate risk with multiple defense strategies. If one layer of defense is insufficient, another layer can prevent a flaw from becoming a vulnerability. |
| 1. Use Effective Quality Assurance Techniques | Effective quality assurance techniques can identify and eliminate vulnerabilities. Independent security reviews can lead to more secure systems. |
| 1. Adopt a Secure Coding Standard | Develop or apply a secure coding standard for your programming language or platform. |

### C/C++ Ten Coding Standards

Complete the coding standards portion of the template according to the Module Three milestone requirements. In Project One, follow the instructions to add a layer of security to the existing coding standards. Please start each standard on a new page, as they may take up more than one page. The first seven coding standards are labeled by category. The last three are blank so you may choose three additional standards. Be sure to label them by category and give them a sequential number for that category. Add compliant and noncompliant sections as needed to each coding standard.

#### Coding Standard 1

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Type** | STD-001-CPP | Choose the proper data types to avoid errors and ensure consistency across platforms. |

| **Noncompliant Code** |
| --- |
|  |
| int count = getCount();  unsigned size = count; // Too risky  short len = size // May cut off data |

| **Compliant Code** |
| --- |
| [Compliant description] |
| uint32\_t count = safeCount();  Size\_t len = static\_cast<size\_t>(count);//uses fixed types to avoid error |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| 2 – Heed Compiler Warnings: Selecting fixed-width types and compiling with high warning levels helps catch implicit conversions. 9 – Use Effective Quality Assurance Techniques: Code reviews and static analysis detect unsafe type usage early. 10 – Adopt a Secure Coding Standard: Following CERT rules for integer and type safety prevents overflow and portability bugs. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Unlikely | Medium | High | 2 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.13 | cert-int30-c, misra-cpp2008-3-9-1 | |  | | --- | |  |  |  | | --- | | Detects unsafe or incorrect integer  type usage | |
| |  | | --- | | Clang-Tidy |  |  | | --- | |  | | 17 | |  | | --- | |  |  |  | | --- | | cppcoreguidelines-pro-type-vararg | | |  | | --- | | Flags risky type conversions and  varargs misuse. |  |  | | --- | |  | |
| SonarQube | 10.4 | cpp:S3649 | Identifies type misuse and potential overflow. |

#### Coding Standard 2

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Data Value** | STD-002-CPP | Numbers that are too small or too large may cause crashes or errors. Always check ranges and use proper types as needed. |

| **Noncompliant Code** |
| --- |
| [Noncompliant description] |
| int a = read\_int();  int b = read\_int();  int bytes = a \* b \* 6; // Could overflow |

| **Compliant Code** |
| --- |
| [Compliant description] |
| bool calc\_bytes(int a, int b, uint64\_t& out) {  if (a <= 0 || b <= 0) return false;  uint64\_t tmp = (uint64\_t)a \* (uint64\_t)b \* 6u;  if (tmp > std::numeric\_limits<size\_t>::max()) return false;  out = tmp;  return true;  } // Prevents overflow by checking before calculations |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| 1 – Validate Input Data: Checking ranges before math prevents overflow from attacker-supplied numbers. 2 – Heed Compiler Warnings: Compiler diagnostics help detect potential overflows. 9 – Use Effective Quality Assurance Techniques: QA and static analysis confirm correct range handling. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Possible | Medium | High | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Cppcheck | 2.13 | |  | | --- | | cert-int32-c, overflowArithmetic |  |  | | --- | |  | | |  | | --- | | Warns about integer overflow and  incorrect value ranges. |  |  | | --- | |  | |
| |  | | --- | | Clang-Tidy |  |  | | --- | |  | | 17 | |  |  |  | | --- | --- | --- | | |  | | --- | |  |  |  | | --- | | cert-int30-c, bugprone-integer-division | |  |  | | --- | |  | | |  |  |  | | --- | --- | --- | | |  | | --- | |  |  |  | | --- | | Detects risky integer math  and division errors. | |  |  | | --- | |  | |
| Coverity | 2024.3 | OVERFLOW\_BEFORE\_WIDEN | Finds overflow issues before type widening. |

#### Coding Standard 3

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **String Correctness** | STD-003-CPP | Dated C string functions are dangerous because they do not check size. Safer options avoid memory issues. |

| **Noncompliant Code** |
| --- |
| [Noncompliant description] |
| char name[32];  strcpy(name, argd[1]); // no size check |

| **Compliant Code** |
| --- |
| [Compliant description] |
| Std::string name;  if (argc > 1){  name = argd[1]; // uses provided argument  } else {  name = “”; //uses an empty string  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| 1 – Validate Input Data: Avoids writing past buffers with unchecked input. 8 – Practice Defense in Depth: Even if one boundary check fails, safer string APIs add a second protection layer. 10 – Adopt a Secure Coding Standard: CERT discourages unsafe C-string functions. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Likely | Low | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Coverity | 2024.3 | STRING\_OVERFLOW, OVERRUN\_STATIC | |  | | --- | | Detects unsafe C-string use and  buffer overruns. |  |  | | --- | |  | |
| |  | | --- | | Clang-Tidy |  |  | | --- | |  | | 17 | |  | | --- | |  |  |  | | --- | | cppcoreguidelines-pro-bounds-  array-to-pointer-decay | | Warns about array-to-pointer decay and unsafe copy. |
| SonarQube | 10.4 | |  | | --- | | cpp:S5753 |  |  | | --- | |  | | Flags unsafe strcpy/strcat and other unsafe string APIs. |

#### Coding Standard 4

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **SQL Injection** | STD-004-CPP | If you build SQL commands by adding user input to strings, harmful code can be introduced. |

| **Noncompliant Code** |
| --- |
| [Noncompliant description] |
| std::string user = req.param("user");  std::string q = "SELECT \* FROM accounts WHERE user='" + user + "';";  db.exec(q); |

| **Compliant Code** |
| --- |
| [Compliant description] |
| auto stmt = db.prepare("SELECT \* FROM accounts WHERE user=?;");  stmt.bind(1, req.param("user"));  auto rows = stmt.exec(); //Removes user input from SQL code |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| 1 – Validate Input Data: Reject or sanitize untrusted SQL parameters. 7 – Sanitize Data Sent to Other Systems: Proper escaping and prepared statements stop injection. 8 – Practice Defense in Depth: Input validation plus parameterized queries and DB permissions, provide layered protection. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Critical | Likely | Low | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| SonarQube | 10.4 | |  | | --- | | cpp:S3649, cpp:S5659 |  |  | | --- | |  | | |  | | --- | | Flags SQL concatenation with user  input. |  |  | | --- | |  | |
| |  | | --- | | Fortify SCA |  |  | | --- | |  | | 24.1 | SQL Injection | |  | | --- | | Detects injection flaws in  database calls. |  |  | | --- | |  | |
| |  | | --- | | OWASP Dependency-Check |  |  | | --- | |  | | 8 | N/a | Finds vulnerable database drivers/libraries. |

#### Coding Standard 5

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Memory Protection** | STD-005-CPP | Forgetting to free memory or freeing it twice can cause crashes or leaks. Smart pointers clean up automatically. |

| **Noncompliant Code** |
| --- |
| [Noncompliant description] |
| Car\* f = new Car();  do\_work();  delete f; // skipped if an error occurs |

| **Compliant Code** |
| --- |
| [Compliant description] |
| auto f = std::make\_unique<Car>();  do\_work(); // cleaned up automatically |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| 2 – Heed Compiler Warnings: Compilers warn on leaks and mismatched allocation. 8 – Practice Defense in Depth: Smart pointers plus sanitizers add multiple safety nets. 9 – Use Effective Quality Assurance Techniques: Static/dynamic tools (ASan, Valgrind) catch leaks and UAF. 10 – Adopt a Secure Coding Standard: CERT MEM rules promote RAII and smart pointers. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Possible | Medium | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| Valgrind | 3.23 | Runtime | |  | | --- | | Detects heap/stack corruption  and UAF. |  |  | | --- | |  | |
| Clang-Tidy | 17 | |  | | --- | |  |  |  | | --- | | modernize-make-unique,  cert-mem50-cpp | | Encourages smart pointers and safe allocation |
| Cppcheck | 2.13 | memleak | Static leak and double free analysis. |

#### Coding Standard 6

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Assertions** | STD-006-CPP | Assertions should be used to check things that are always true in the program. They should not check user input, if statements should be used instead |

| **Noncompliant Code** |
| --- |
| [Noncompliant description] |
| int age = read\_age();  assert(age >= 0 && age <= 130); |

| **Compliant Code** |
| --- |
| [Compliant description] |
| int age = read\_age();  if (age < 0 || age > 130) return -1;// check  assert(age >= 0); |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| 3 – Architect and Design for Security Policies: Design code so assertions guard only internal invariants. 9 – Use Effective Quality Assurance Techniques: Proper assert use found during code review/testing. 10 – Adopt a Secure Coding Standard: CERT MSC guidelines state asserts should not validate external input. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Medium | Possible | Low | Medium | 3 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| |  | | --- | | Clang-Tidy |  |  | | --- | |  | | 17 | |  | | --- | | cert-msc30-c |  |  | | --- | |  | | |  | | --- | | Warns on misuse of assert with  external data. |  |  | | --- | |  | |
| Cppcheck | 2.13 | |  | | --- | | assertWithSideEffect |  |  | | --- | |  | | |  | | --- | | Detects unsafe assert expressions. |  |  | | --- | |  | |
| SonarQube | 10.4 | |  | | --- | | cpp:S5411 |  |  | | --- | |  | | Flags assert use on untrusted input. |

#### Coding Standard 7

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| **Exceptions** | STD-007-CPP | Catch exceptions by reference so no information is lost. Do not throw exceptions from destructors, as that can crash your program. |

| **Noncompliant Code** |
| --- |
| [Noncompliant description] |
| struct Handle {  ~Handle(){ throw std::runtime\_error("oops"); }  };  try {Handle h;} catch (std::exception e) {} |

| **Compliant Code** |
| --- |
| [Compliant description] |
| struct Handle{~Handle() noexcept {} };  try {  might\_fail();  } catch (const std::exception& ex) {  log(ex.what());  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| 3 – Architect and Design for Security Policies: Exception handling is part of secure architecture. 8 – Practice Defense in Depth: Defensive exception safety reduces cascading failures. 9 – Use Effective Quality Assurance Techniques: Reviews and tools verify noexcept destructors and catch-by-ref. 10 – Adopt a Secure Coding Standard: CERT ERR rules guide safe throwing/catching. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Possible | Medium | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| |  | | --- | | Clang-Tidy |  |  | | --- | |  | | 17 | |  | | --- | | cert-err60-cpp |  |  | | --- | |  | | |  | | --- | | Detects throws in destructors  & bad exception safety. |  |  | | --- | |  | |
| |  | | --- | | SonarQube |  |  | | --- | |  | | 10.4 | |  | | --- | | cpp:S3623 |  |  | | --- | |  | | |  | | --- | | Warns about catching  exceptions by value. |  |  | | --- | |  | |
| Coverity | 2024.3 | |  | | --- | | EXCEPT\_THROW\_IN\_DTOR |  |  | | --- | |  | | Finds destructor throw problems. |

#### Coding Standard 8

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Data Logging | STD-008-CPP | Never print sensitive or personal data in logs. Logs can be stolen or read by others. |

| **Noncompliant Code** |
| --- |
| [Noncompliant description] |
| log("login user=", user, " pw=", pw, " token=", token); |

| **Compliant Code** |
| --- |
| [Compliant description] |
| log("login user=", user, " pw=<hidden> token=<hidden>"); //Hidden |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| 5 – Default Deny: Deny by default logging of sensitive data. 8 – Practice Defense in Depth: Hidden tokens in logs plus secure transport protect secrets if logs leak. 9 – Use Effective Quality Assurance Techniques: Security reviews ensure logs never expose PII. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Possible | Low | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| |  | | --- | | Fortify SCA |  |  | | --- | |  | | 24.1 | |  | | --- | | PrivacyViolation, LogForging |  |  | | --- | |  | | |  | | --- | | Detects PII/secret leakage in logs. |  |  | | --- | |  | |
| SonarQube | 10.4 | |  | | --- | | cpp:S5144 |  |  | | --- | |  | | |  | | --- | | Flags sensitive info in log statements. |  |  | | --- | |  | |
| |  | | --- | | Checkmarx SAST |  |  | | --- | |  | | 9 | Sensitive\_Data\_Logging | |  | | --- | | Detects insecure logging of credentials/tokens. | |

#### Coding Standard 9

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Pointer Safety | STD-009-CPP | Storing or returning pointers incorrectly can cause crashes. Use smart pointers for ownership, and never return pointers to local variables. |

| **Noncompliant Code** |
| --- |
| [Noncompliant description] |
| int\* getAnswer() {  int x = 42;  return &x; // dangling pointer  } |

| **Compliant Code** |
| --- |
| [Compliant description] |
| int getAnswer() { // no ownership issues  return 42;  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| 2 – Heed Compiler Warnings: Compilers and static analyzers catch dangling pointers. 8 – Practice Defense in Depth: Smart pointers plus runtime sanitizers add multiple protections. 9 – Use Effective Quality Assurance Techniques: QA tools (ASan/UBSan) detect invalid pointer usage. 10 – Adopt a Secure Coding Standard: CERT MEM rules prevent returning addresses of locals. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| High | Possible | Medium | High | 4 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| |  | | --- | | Clang Static  Analyzer |  |  | | --- | |  | | 17 | |  | | --- | |  |  |  | | --- | | DanglingPtr, ReturnStackAddress | | |  | | --- | | Detects dangling and stack  address returns. |  |  | | --- | |  | |
| Cppcheck | 2.13 | |  | | --- | | nullPointer, memleak |  |  | | --- | |  | | |  | | --- | | Pointer misuse and leak detection. |  |  | | --- | |  | |
| |  | | --- | | AddressSanitizer  /UBSan |  |  | | --- | |  | | |  | | --- | | LLVM/  Clang 17 |  |  | | --- | |  | | runtime | |  | | --- | | Detects invalid memory/pointer access at runtime. | |

#### Coding Standard 10

| **Coding Standard** | **Label** | **Name of Standard** |
| --- | --- | --- |
| Input Validation | STD-010-CPP | Always treat input as unsafe until validated. Checking type, size, and range prevents buffer overflow, crashes, and injection attacks. |

| **Noncompliant Code** |
| --- |
| [Noncompliant description] |
| int age;  std::cin >> age; // no validation |

| **Compliant Code** |
| --- |
| [Compliant description] |
| int age;  if (!(std::cin >> age) || age < 0 || age > 130) {  std::cerr << "Invalid input\n";  } |

**Note: Stop here for the milestone. Complete this section for Project One in Module Six.**

| 1 – Validate Input Data: Treat all input as untrusted until checked. 5 – Default Deny: Reject invalid or unexpected input by default. 7 – Sanitize Data Sent to Other Systems: Only forward validated/escaped data. 8 – Practice Defense in Depth: Combine type/length checks, whitelisting, and runtime guards. |
| --- |

**Threat Level**

| **Severity** | **Likelihood** | **Remediation Cost** | **Priority** | **Level** |
| --- | --- | --- | --- | --- |
| Critical | Likely | Low | High | 5 |

**Automation**

| **Tool** | **Version** | **Checker** | **Description Tool** |
| --- | --- | --- | --- |
| |  | | --- | | SonarQube |  |  | | --- | |  | | 10.4 | |  | | --- | | cpp:S3649, cpp:S5773 |  |  | | --- | |  | | |  | | --- | | Flags tainted/unchecked input. |  |  | | --- | |  | |
| |  | | --- | | Fortify SCA |  |  | | --- | |  | | 24.1 | |  | | --- | | Input Validation & Representation |  |  | | --- | |  | | |  | | --- | | Detects missing validation and  sanitization. |  |  | | --- | |  | |
| |  | | --- | | OWASP ZAP |  |  | | --- | |  | | 2.15 | |  | | --- | | Active Scan |  |  | | --- | |  | | |  | | --- | | Fuzzes input fields during pre-prod testing. | |

Defense-in-Depth Illustration

This illustration provides a visual representation of the defense-in-depth best practice of layered security.



## Project One

There are seven steps outlined below that align with the elements you will be graded on in the accompanying rubric. When you complete these steps, you will have finished the security policy.

### Revise the C/C++ Standards

You completed one of these tables for each of your standards in the Module Three milestone. In Project One, add revisions to improve the explanation and examples as needed. Add rows to accommodate additional examples of compliant and noncompliant code. Coding standards begin on the security policy.

### Risk Assessment

Complete this section on the coding standards tables. Enter high, medium, or low for each of the headers, then rate it overall using a scale from 1 to 5, 5 being the greatest threat. You will address each of the seven policy standards. Fill in the columns of severity, likelihood, remediation cost, priority, and level using the values provided in the appendix.

### Automated Detection

Complete this section of each table on the coding standards to show the tools that may be used to detect issues. Provide the tool name, version, checker, and description. List one or more tools that can automatically detect this issue and its version number, name of the rule or check (preferably with link), and any relevant comments or description—if any. This table ties to a specific C++ coding standard.

### Automation

Provide a written explanation using the image provided.



Automation will be used for the enforcement of and compliance to the standards defined in this policy. Green Pace already has a well-established DevOps process and infrastructure. Define guidance on where and how to modify the existing DevOps process to automate enforcement of the standards in this policy. Use the DevSecOps diagram and provide an explanation using that diagram as context.

Automation within the Green Pace DevSecOps pipeline is implemented to ensure consistent enforcement of secure coding standards and early identification of faults and defects. Automated analysis shall be integrated at key stages of the software development lifecycle. During the create and verify stages, static analysis will be used to detect violations of the coding standards. These checks should occur automatically to ensure poor coding does not progress into later development stages.

During pre-production, controlled unit testing should be performed to evaluate error handling and input validation. Pipelines should automatically stop code that fails to meet security thresholds. During the release and prevent stages, integrity validation and policy conformity checks should confirm that no code reaches deployment without passing this secure development policy.

### Summary of Risk Assessments

Consolidate all risk assessments into one table including both coding and systems standards, ordered by standard number.

| Rule | Severity | Likelihood | Remediation Cost | Priority | Level |
| --- | --- | --- | --- | --- | --- |
| STD-001-CPP | High | Unlikely | Medium | High | 2 |
| STD-002-CPP | High | Possible | Medium | High | 3 |
| STD-003-CPP | High | Likely | Low | High | 5 |
| STD-004-CPP | Critical | Likely | Low | High | 5 |
| STD-005-CPP | High | Possible | Medium | High | 4 |
| STD-006-CPP | Medium | Possible | Low | Medium | 3 |
| STD-007-CPP | High | Possible | Medium | High | 4 |
| STD-008-CPP | High | Possible | Low | High | 4 |
| STD-009-CPP | High | Possible | Medium | High | 4 |
| STD-010-CPP | Critical | Likely | Low | High | 5 |

### Create Policies for Encryption and Triple A

Include all three types of encryption (in flight, at rest, and in use) and each of the three elements of the Triple-A framework using the tables provided***.***

* 1. Explain each type of encryption, how it is used, and why and when the policy applies.
  2. Explain each type of Triple-A framework strategy, how it is used, and why and when the policy applies.

Write policies for each and explain what it is, how it should be applied in practice, and why it should be used.

| 1. **Encryption** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Encryption at rest | Encryption at rest protects data that is stored on servers, databases, and backups using cryptographic algorithms. All data must be stored using industry standards to prevent unauthorized access. |
| Encryption in flight | Encryption in flight ensures data transmissions are protected from tampering or interception. Transport layer security protocols should be enforced for all internal or external communications. |
| Encryption in use | Encryption in use applies to data currently in use. Sensitive data must be protected through secure memory handling and process isolation. This policy minimizes risk from runtime exploits and unauthorized observation during execution. |

| 1. **Triple-A Framework\*** | **Explain what it is and how and why the policy applies.** |
| --- | --- |
| Authentication | Authentication verifies users or system identities before granting access to platforms or code repositories. Strong credentials and multi-factor authentication shall be used for all logins. |
| Authorization | Authorization controls the level of access each user can exercise. Least privilege principles must be applied to users, developers, and only have permissions for their assigned duties. Role-based access control and default deny shall be enforced. |
| Accounting | Accounting ensures all security-related events are logged in tamper-proof audit logs. These records ensure accountability, aid investigations, and ensure compliance. Audit logs must be preserved in accordance to retention and review policies. |

**\***Use this checklist for the Triple A to be sure you include these elements in your policy:

* User logins
* Changes to the database
* Addition of new users
* User level of access
* Files accessed by users

### Map the Principles

Map the principles to each of the standards, and provide a justification for the connection between the two. In the Module Three milestone, you added definitions for each of the 10 principles provided. Now it’s time to connect the standards to principles to show how they are supported by principles. You may have more than one principle for each standard, and the principles may be used more than once. Principles are numbered 1 through 10. You will list the number or numbers that apply to each standard, then explain how each of these principles supports the standard. This exercise demonstrates that you have based your security policy on widely accepted principles. Linking principles to standards is a best practice.

**NOTE:** Green Pace has already successfully implemented the following:

* Operating system logs
* Firewall logs
* Anti-malware logs

The only item you must complete beyond this point is the Policy Version History table.

## Audit Controls and Management

Every software development effort must be able to provide evidence of compliance for each software deployed into any Green Pace managed environment.

Evidence will include the following:

* Code compliance to standards
* Well-documented access-control strategies, with sampled evidence of compliance
* Well-documented data-control standards defining the expected security posture of data at rest, in flight, and in use
* Historical evidence of sustained practice (emails, logs, audits, meeting notes)

## Enforcement

The office of the chief information security officer (OCISO) will enforce awareness and compliance of this policy, producing reports for the risk management committee (RMC) to review monthly. Every system deployed in any environment operated by Green Pace is expected to be in compliance with this policy at all times.

Staff members, consultants, or employees found in violation of this policy will be subject to disciplinary action, up to and including termination.

## Exceptions Process

Any exception to the standards in this policy must be requested in writing with the following information:

* Business or technical rationale
* Risk impact analysis
* Risk mitigation analysis
* Plan to come into compliance
* Date for when the plan to come into compliance will be completed

Approval for any exception must be granted by chief information officer (CIO) and the chief information security officer (CISO) or their appointed delegates of officer level.

Exceptions will remain on file with the office of the CISO, which will administer and govern compliance.

## Distribution

This policy is to be distributed to all Green Pace IT staff annually. All IT staff will need to certify acceptance and awareness of this policy annually.

## Policy Change Control

This policy will be automatically reviewed annually, no later than 365 days from the last revision date. Further, it will be reviewed in response to regulatory or compliance changes, and on demand as determined by the OCISO.

## Policy Version History

| Version | Date | Description | Edited By | Approved By |
| --- | --- | --- | --- | --- |
| 1.0 | 08/05/2020 | Initial Template | David Buksbaum |  |
| 1.1 | 10/12/2025 | Policy update | Chris Bridges | SNHU |

## Appendix A Lookups

### Approved C/C++ Language Acronyms

| Language | Acronym |
| --- | --- |
| C++ | CPP |
| C | CLG |
| Java | JAV |